From Legion to Avaki: The Persistence of Vision[*]

Andrew S. Grimshaw
Anand Natrajan
Marty A. Humphrey
Michael J. Lewis
Anh Nguyen-Tuong
John F. Karpovich
Mark M. Morgan
Adam J. Ferrari

Abstract:

Grids have metamorphosed from academic projects to commercial ventures. Avaki, a leading commercial vendor of Grids, has its roots in Legion, a Grid project at the University of Virginia begun in 1993. In this chapter, we present fundamental challenges and requirements for Grid architectures that we believe are universal, our architectural philosophy in addressing those requirements, an overview of Legion as used in production systems and a synopsis of the Legion architecture and implementation. We also describe the history of the transformation from Legion – an academic, research project – to Avaki, a commercially supported, marketed product. Several of the design principles as well as the vision underlying Legion have continued to be employed in Avaki. As a product sold to customers, Avaki has been made more robust, more easily manageable and easier to configure than Legion, at the expense of eliminating some features and tools that are of less immediate use to customers. Finally, we place Legion in the context of OGSI, a standards effort underway in Global Grid Forum.

[*] This work partially supported by DARPA (Navy) contract #N66001-96-C-8527, DOE grant DE-FG02-96ER25290, DOE contract Sandia LD-9391, Logicon (for the DoD HPCMOD/PET program) DAHC 94-96-C-0008, DOE D459000-16-3C, DARPA (GA) SC H607305A, NSF-NGS EIA-9974968, NSF-NPACI ASC-96-10920, and a grant from NASA-IPG.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **2006**	2. REPORT TYPE	3. DATES COVERED **00-00-2006 to 00-00-2006**
4. TITLE AND SUBTITLE **From Legion to Avaki: The Persistence of Vision**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Virginia,Department of Computer Science,151 Engineer's Way,Cahrlottesville,VA,22094-4740**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES **The original document contains color images.**		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**30**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

1 Grids Are Here

In 1994, we outlined our vision for wide-area distributed computing [9]:

> For over thirty years science fiction writers have spun yarns featuring worldwide networks of interconnected computers that behave as a single entity. Until recently such science fiction fantasies have been just that. Technological changes are now occurring which may expand computational power in the same way that the invention of desk top calculators and personal computers did. In the near future computationally demanding applications will no longer be executed primarily on supercomputers and single workstations using local data sources. Instead enterprise-wide systems, and someday nationwide systems, will be used that consist of workstations, vector supercomputers, and parallel supercomputers connected by local and wide area networks. Users will be presented the illusion of a single, very powerful computer, rather than a collection of disparate machines. The system will schedule application components on processors, manage data transfer, and provide communication and synchronization in such a manner as to dramatically improve application performance. Further, boundaries between computers will be invisible, as will the location of data and the failure of processors.

The future is now; after almost a decade of research and development by the Grid community we see Grids (then called metasystems [25]) being deployed around the world both in academic settings, and more tellingly, for production commercial use.

This chapter describes one of the major Grid projects of the last decade, Legion, from its roots as an academic Grid project [10][15][16] to its current status as the only commercial complete Grid offering, Avaki, marketed by a Cambridge, Massachusetts company called AVAKI Corporation. We begin with a discussion of the fundamental requirements for any Grid architecture. These fundamental requirements continue to guide the evolution of our Grid software. We then present some of the principles and philosophy underlying the design of Legion. Next, we present briefly what a Legion grid looks like to administrators and users. We introduce some of the architectural features of Legion and delve slightly deeper into implementation in order to give an intuitive understanding of grids and Legion. Detailed technical descriptions abound in the literature [12][17][4][5][13][1][14]. We then present a brief history of Legion and Avaki in order to place the preceding discussion in context. We conclude with a look at the future and how Legion and Avaki fit in with emerging standards such as OGSI [8].

2 Grid Architecture Requirements

What use is a Grid? What is required of a Grid? Before we answer these questions, let us step back and define what is a grid and what are its essential attributes.

Our definition, and indeed a popular definition, is: A Grid *system* is a collection of distributed resources connected by a network. A Grid system, also called a *Grid*, gathers resources – desktop and hand-held hosts, devices with embedded processing resources such as digital cameras and phones or tera-scale supercomputers – and makes them accessible to users and applications in order to reduce overhead and accelerate projects. A Grid *application* can be defined as an application that operates in a Grid environment or is "on" a Grid system. Grid *system software* (or middleware), is software that facilitates writing Grid applications and manages the underlying Grid infrastructure.

The resources in a Grid typically share at least some of the following characteristics:

- They are numerous.

- They are owned and managed by different, potentially mutually-distrustful organizations and individuals.
- They are potentially faulty.
- They have different security requirements and policies.
- They are heterogeneous, i.e., they have different CPU architectures, are running different operating systems, and have different amounts of memory and disk.
- They are connected by heterogeneous, multilevel networks.
- They have different resource management policies.
- They are likely to be geographically-separated (on a campus, in an enterprise, on a continent).

A Grid enables users to collaborate securely by sharing *processing, applications* and *data* across systems with the above characteristics in order to facilitate collaboration, faster application execution and easier access to data. More concretely this means being able to:

Find and share data. When users need access to data on other systems or networks, they should simply be able to access it like data on their own system. System boundaries that are not useful should be invisible to users who have been granted legitimate access to the information.

Find and share applications. The leading edge of development, engineering and research efforts consists of custom applications – permanent or experimental, new or legacy, public-domain or proprietary. Each application has its own requirements. Why should application users have to jump through hoops to get applications together with the data sets needed for analysis?

Share computing resources. It sounds very simple – one group has computing cycles, some colleagues in another group need them. The first group should be able to grant access to its own computing power without compromising the rest of the network.

Grid computing is in many ways a novel way to construct applications. It has received a significant amount of recent press attention and been heralded as the next wave in computing. However, under the guises of “peer-to-peer systems”, “metasystems” and “distributed systems”, Grid computing requirements and the tools to meet these requirements have been under development for decades. Grid computing requirements address the issues that frequently confront a developer trying to construct applications for a grid. The novelty in grids is that these requirements are addressed by the grid infrastructure in order to reduce the burden on the application developer. The requirements are:

- ***Security.*** Security covers a gamut of issues, including authentication, data integrity, authorization (access control) and auditing. If Grids are to be accepted by corporate and government IT departments, a wide range of security concerns must be addressed. Security mechanisms must be integral to applications and capable of supporting diverse policies. Furthermore, we believe that security must be firmly built in from the beginning. Trying to patch security in as an afterthought (as some systems are attempting today) is a fundamentally flawed approach. We also believe that no single security policy is perfect for all users and organizations. Therefore, a Grid system must have mechanisms that allow users and resource owners to select policies that fit particular security and performance needs, as well as meet local administrative requirements.
- ***Global name space.*** The lack of a global name space for accessing data and resources is one of the most significant obstacles to wide-area distributed and parallel processing. The current multitude of disjoint name spaces greatly impedes developing applications that span sites. All Grid objects must be able to access

(subject to security constraints) any other Grid object *transparently* without regard to location or replication.

- ***Fault tolerance.*** Failure in large-scale Grid systems is and will be a fact of life. Hosts, networks, disks and applications frequently fail, restart, disappear and behave otherwise unexpectedly. Forcing the programmer to predict and handle all of these failures significantly increases the difficulty of writing reliable applications. Fault-tolerant computing is a known, very difficult problem. Nonetheless it must be addressed or businesses and researchers will not entrust their data to Grid computing.
- ***Accommodating heterogeneity.*** A Grid system must support interoperability between heterogeneous hardware and software platforms. Ideally, a running application should be able to migrate from platform to platform if necessary. At a bare minimum, components running on different platforms must be able to communicate transparently.
- ***Binary management.*** The underlying system should keep track of executables and libraries, knowing which ones are current, which ones are used with which persistent states, where they have been installed and where upgrades should be installed. These tasks reduce the burden on the programmer.
- ***Multi-language support.*** In 1970s, the joke was "I don't know what language they'll be using in the year 2000, but it'll be called Fortran." Fortran has lasted over 40 years, and C almost 30. Diverse languages will always be used and legacy applications will need support.
- ***Scalability.*** There are over 400 million computers in the world today and over 100 million network-attached devices (including computers). Scalability is clearly a critical necessity. Any architecture relying on centralized resources is doomed to failure. A successful Grid architecture must strictly adhere to the distributed systems principle: the service demanded of any given component must be independent of the number of components in the system. In other words, the service load on any given component must not increase as the number of components increases.
- ***Persistence.*** I/O and the ability to read and write persistent data are critical in order to communicate between applications and to save data. However, the current files/file libraries paradigm should be supported, since it is familiar to programmers.
- ***Extensibility.*** Grid systems must be flexible enough to satisfy current user demands and unanticipated future needs. Therefore, we feel that mechanism and policy must be realized by replaceable and extensible components, including (and especially) core system components. This model facilitates development of improved implementations that provide value-added services or site-specific policies while enabling the system to adapt over time to a changing hardware and user environment.
- ***Site autonomy.*** Grid systems will be composed of resources owned by many organizations, each of which desire to retain control over their own resources. For each resource the owner must be able to limit or deny use by particular users, specify when it can be used, etc. Sites must also be able to choose or rewrite an implementation of each Legion component as best suits their needs. A given site may trust the security mechanisms of one particular implementation over those of another so it should be freely able to use that implementation.
- ***Complexity management***. Finally, but importantly, complexity management is one of the biggest challenges in large scale Grid systems. In the absence of system support, the application programmer is faced with a confusing array of decisions. Complexity exists in multiple dimensions: heterogeneity in policies for resource

usage and security, a range of different failure modes and different availability requirements, disjoint namespaces and identity spaces, and the sheer number of components. For example, professionals who are not IT experts should not have to remember the details of five or six different file systems and directory hierarchies (not to mention multiple user names and passwords) in order to access the files they use on a regular basis. Thus, providing the programmer and system administrator with clean abstractions is critical to reducing the cognitive burden.

Solving these requirements is the task of a Grid infrastructure. A architecture for a Grid based on well-thought principles is required in order to address each of these requirements. In the next section, we discuss the principles underlying the design of one Grid system, namely, Legion.

3 Legion Principles and Philosophy

Legion is a Grid architecture as well as an operational infrastructure under development since 1993 at the University of Virginia. The architecture addresses the requirements of the previous section and builds on lessons learned from earlier systems. We defer a discussion of the history of Legion and its transition to a commercial product named Avaki to Section 7. Here, we focus on the design principles and philosophy of Legion, which can be encapsulated in the following "rules":

- ***Provide a single-system view.*** With today's operating systems we can maintain the illusion that our local area network is a single computing resource. But once we move beyond the local network or cluster to a geographically-dispersed group of sites, perhaps consisting of several different types of platforms, the illusion breaks down. Researchers, engineers and product development specialists (most of whom do not want to be experts in computer technology) must request access through the appropriate gatekeepers, manage multiple passwords, remember multiple protocols for interaction, keep track of where everything is located, and be aware of specific platform-dependent limitations (e.g., this file is too big to copy or to transfer to one's system; that application runs only on a certain type of computer). Re-creating the illusion of single computing resource for heterogeneous, distributed resources reduces the complexity of the overall system and provides a single namespace.
- ***Provide transparency as a means of hiding detail.*** Grid systems should support the traditional distributed system transparencies: access, location, heterogeneity, failure, migration, replication, scaling, concurrency and behavior [17]. For example, users and programmers should not have to know where an object is located in order to use it (access, location and migration transparency), nor should they need to know that a component across the country failed – they want the system to recover automatically and complete the desired task (failure transparency). This is the traditional way to mask various aspects of the underlying system. Transparency addresses fault-tolerance and complexity.
- ***Provide flexible semantics.*** Our overall objective was a Grid architecture that is suitable to as many users and purposes as possible. A rigid system design in which policies are limited, trade-off decisions are pre-selected, or all semantics are pre-determined and hard-coded would not achieve this goal. Indeed, if we dictated a single system-wide solution to almost any of the technical objectives outlined above, we would preclude large classes of potential users and uses. Therefore, Legion allows users and programmers as much flexibility as possible in their applications' semantics, resisting the temptation to dictate solutions. Whenever possible, users can select both the *kind* and the *level* of functionality and choose their own trade-offs between function and cost. This philosophy is manifested in the system architecture. The Legion object model specifies the functionality but not the implementation of the

system's core objects; the core system therefore consists of extensible, replaceable components. Legion provides default implementations of the core objects, although users are not obligated to use them. Instead, we encourage users to select or construct object implementations that answer their specific needs.

- ***By default the user should not have to think.*** In general there are four classes of users of grids: end-users of applications, applications developers, system administrators and managers who are trying to accomplish some mission with the available resources. We believe that users want to focus on their jobs, i.e., their applications, and not on the underlying Grid plumbing and infrastructure. Thus, for example, to run an application a user may type

  legion_run my_application my_data

  at the command shell. The Grid should then take care of all of the messy details such as finding an appropriate host on which to execute the application, moving data and executables around, etc. Of course, the user may optionally be aware and specify or override certain behaviors, for example, specify an architecture on which to run the job, or name a specific machine or set of machines, or even replace the default scheduler.
- ***Reduce "activation energy".*** One of the typical problems in technology adoption is getting users to use it. If it is difficult to shift to a new technology then users will tend not to take the effort to try it unless their need is immediate and extremely compelling. This is not a problem unique to Grids – it is human nature. Therefore, one of our most important goals is to make using the technology easy. Using an analogy from chemistry, we keep the activation energy of adoption as low as possible. Thus, users can easily and readily realize the benefit of using Grids – and get the reaction going – creating a self-sustaining spread of Grid usage throughout the organization. This principle manifests itself in features such as "no recompilation" for applications to be ported to a Grid, and support for mapping a Grid to a local operating system file system. Another variant of this concept is the motto "no play, no pay". The basic idea is that if you do not need a feature, e.g., encrypted data streams, fault resilient files or strong access control, you should not have to pay the overhead of using it.
- ***Do not change host operating systems.*** Organizations will not permit their machines to be used if their operating systems must be replaced. Our experience with Mentat [11] indicates, though, that building a Grid on top of host operating systems is a viable approach.
- ***Do not change network interfaces.*** Just as we must accommodate existing operating systems, we assume that we cannot change the network resources or the protocols in use.
- ***Do not require Grids to run in privileged mode.*** To protect their objects and resources, Grid users and sites will require Grid software to run with the lowest possible privileges.

Although we focus primarily on technical issues in this chapter, we recognize that there are also important political, sociological and economic challenges in developing and deploying Grids, such as developing a scheme to encourage the participation of resource-rich sites while discouraging free-riding by others. Indeed, politics can often overwhelm technical issues.

4 Using Legion in Day-to-Day Operations

Legion is comprehensive Grid software that enables efficient, effective and secure sharing of data, applications and computing power. It addresses the technical and administrative challenges faced by organizations such as research, development and engineering groups with computing resources in disparate locations, on heterogeneous platforms and under multiple administrative jurisdictions. Since Legion enables these diverse, distributed resources to be treated as a single virtual operating environment with a single file structure, it drastically reduces the overhead of sharing data, executing applications and utilizing available computing power regardless of location or platform. The central feature in Legion is the single global name space. Everything in Legion has a name: hosts, files, directories, groups for security, schedulers, applications, etc. The same name is used regardless of where the name is used and regardless of where the named object resides at any given point in time.

In this and the following sections, we use the term "Legion" to mean both the academic project at the University of Virginia as well as the commercial product, Avaki, distributed by AVAKI Corp.

Legion helps organizations create a *compute Grid*, allowing processing power to be shared, as well as a *data Grid*, a virtual single set of files that can be accessed without regard to location or platform. Fundamentally, a compute Grid and a data Grid are the same product – the distinction is solely for the purposes of exposition. Legion's unique approach maintains the security of network resources while reducing disruption to current operations. By increasing sharing, reducing overhead and implementing with low disruption, Legion delivers important efficiencies that translate to reduced cost.

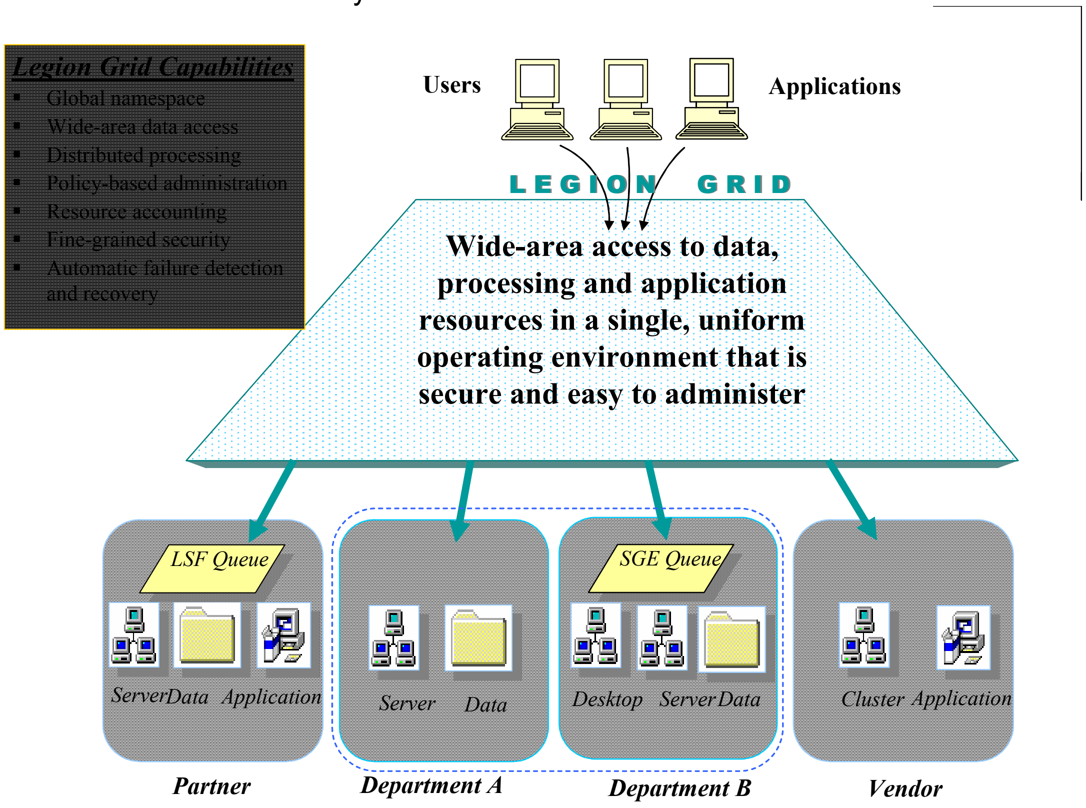
We start with a somewhat typical scenario and how it might appear to the end-user. Suppose we have a small Grid as shown below with four sites, two different departments in one company, a partner site and a vendor site. Two sites are using load management systems; the partner is using Platform Computing™ Load Sharing Facility (LSF) software and one department is using Sun™ Grid Engine (SGE). We will assume that there is a mix of hardware in the Grid, e.g., Linux hosts, Solaris hosts, AIX hosts, Windows 2000 and Tru64 Unix. Finally, there is data of interest at three different sites. A user then sits down at a terminal, authenticates to Legion (logs in) and runs the command

legion_run my_application my_data

Legion will then *by default*, determine the binaries available, find and select a host on which to execute *my_application*, manage the secure transport of credentials, interact with the local operating environment on the selected host (perhaps an SGE™ queue), create accounting records, check to see if the current version of the application has been installed (and if not install it), move all of the data around as necessary, and return the results to the user. The user does not need to know where the application resides, where the execution occurs, where the file *my_data* is physically located, nor any other of the myriad details of what it takes to execute the application. Of course, the user may choose to be aware of, and specify or override, certain behaviors, for example, specify an architecture on which to run the job, or name a specific machine or set of machines, or even replace the default scheduler. In this example the user exploits key features:

- Global name space. Everything she specifies is in terms of a global name space that names everything: processors, applications, queues, data files and directories. The same name is used regardless of the location of the user of the name or the location of the named entity.
- Wide-area access to data. All of the named entities, including files, are mapped into the local file system directory structure of her workstation, making access to the Grid transparent.

- Access to distributed and heterogeneous computing resources. Legion keeps track of binary availability and the current version.
- Single sign-on. The user need not keep track of multiple accounts at different sites. Indeed, Legion supports policies that do not require a local account at a site to access data or execute applications, as well as policies that require local accounts.
- Policy-based administration of the resource base. Administration is as important as application execution.
- Accounting both for resource usage information and auditing purposes. Legion monitors and maintains a RDBMS with accounting information such as who used what application on what host, starting when and how much was used.
- Fine-grained security that protects both her resources and those of others.
- Failure detection and recovery.



4.1 Creating and Administering a Legion Grid

Legion enables organizations to collect resources – applications, computing power and data – to be used as a single virtual operating environment. This set of shared resources is called a Legion Grid. A Legion Grid can represent resources from homogeneous platforms at a single site within a single department, as well as resources from multiple sites, heterogeneous platforms and separate administrative domains.

Legion ensures secure access to resources on the Grid. Files on participating computers become part of the Grid only when they are *shared*, or explicitly made available to the Grid. Further, even when shared, Legion's fine-grained access control is used to prevent unauthorized access. Any subset of resources can be shared, for example, only the processing power or only certain files or directories. Resources that have not been shared are not visible to Grid users. By the same token, a user of an individual computer or network that participates in the Grid is not automatically a Grid user and does not automatically have access to Grid files. Only users who have explicitly been granted access can take advantage of the shared resources. Local administrators may retain control over who can use their computers, at what time of day and under which load conditions. Local resource owners control access to their resources.

Once a Grid is created, users can think of it as one computer with one directory structure and one batch processing protocol. They need not know where individual files are located physically, on what platform type or under which security domain. A Legion Grid can be administered in different ways, depending on the needs of the organization.

1. **As a single administrative domain.** When all resources on the Grid are owned or controlled by a single department or division, it is sometimes convenient to administer them centrally. The administrator controls which resources are made available to the Grid and grants access to resources. In this case, there may still be separate administrators at the different sites who are responsible for routine maintenance of the local systems.
2. **As a federation of multiple administrative domains.** When resources are part of multiple administrative domains, as is the case with multiple divisions or companies cooperating on a project, more control is left to administrators of the local networks. They each define which of their resources are made available to the Grid and who has access. In this case, a team responsible for the collaboration would provide any necessary information to the system administrators, and would be responsible for the initial establishment of the Grid.

With Legion, there is little or no intrinsic need for central administration of a Grid. Resource owners are administrators for their own resources and can define who has access to them. Initially administrators cooperate in order to create the Grid; after that, it is a simple matter of which management controls the organization wants to put in place. In addition, Legion provides features specifically for the convenience of administrators who want to track queues and processing across the Grid. With Legion, they can:

- Monitor local and remote load information on all systems for CPU use, idle time, load average and other factors from any machine on the grid.
- Add resources to queues or remove them without system interruption, and dynamically configure resources based on policies and schedules.
- Log warnings and error messages and filter them by severity.
- Collect all resource usage information down to the user, file, application or project level, enabling Grid-wide accounting.
- Create scripts of Legion commands to automate common administrative tasks.

4.2 Legion Data Grid

Data access is critical for any application or organization. A Legion Data Grid [25] greatly simplifies the process of interacting with resources in multiple locations, on multiple platforms or under multiple administrative domains. Users access files by name – typically a pathname in the Legion virtual directory. *There is no need to know the physical location of the files.*

There are two basic concepts to understand in the Legion Data Grid – how the data is accessed, and how the data is included into the Grid.

4.2.1 Data Access

Data access is through one of three mechanisms: a Legion-aware NFS server called a Data Access Point (hereafter a *DAP*), a set of command line utilities or Legion I/O libraries that mimic the C *stdio* libraries.

DAP Access

The DAP provides a standards-based mechanism to access a Legion Data Grid. It is a commonly-used mechanism to access data in a Data Grid. The DAP is a server that responds to NFS 2.0/3.0 protocols and interacts with the Legion system. When an NFS client on a host mounts a DAP it effectively maps the Legion global name space into the local host file system, providing completely transparent access to data throughout the Grid without even installing Legion software.

However, the DAP is not a typical NFS server. First, it has no actual disk or file system behind it – it interacts with a set of resources that may be distributed, be owned by multiple organizations, be behind firewalls, etc. Second, the DAP supports the Legion security mechanisms – access control is with signed credentials, and interactions with the data Grid can be encrypted. Third, the DAP caches data aggressively, using configurable local memory and disk caches to avoid wide-area network access. Further, the DAP can be modified to exploit semantic data that can be carried in the meta-data of a file object, such as "cacheable", "cacheable until" or "coherence window size". In effect, DAP provides a highly secure, wide-area NFS.

To avoid the rather obvious hot-spot of a single DAP at each site, Legion encourages deploying more than one DAP per site. There are two extremes: one DAP per site, and one DAP per host. Besides the obvious tradeoff between scalability and the shared cache effects of these two extremes, there is also an added security benefit of having one DAP per host. NFS traffic between the client and the DAP, typically unencrypted, can be restricted to one host. The DAP can be configured to only accept requests from local host, eliminating the classic NFS security attacks through network spoofing.

Command Line Access

A Legion Data Grid can be accessed using a set of command line tools that mimic the Unix file system commands such as *ls*, *cat*, etc. The Legion analogues are *legion_ls*, *legion_cat*, etc. The Unix-like syntax is intended to mask the complexity of remote data access by presenting familiar semantics to users.

I/O Libraries

Legion provides a set of I/O libraries that mimic the *stdio* libraries. Functions such as *open* and *fread* have analogues such as *BasicFiles_open* and *BasicFiles_fread*. The libraries are used by applications that need stricter coherence semantics than offered by NFS access. The library functions operate directly on the relevant file or directory object rather than operating via the DAP caches.

4.2.2 Data Inclusion

Data inclusion is through one of three mechanisms: a "copy" mechanism whereby a copy of the file is made in the grid, a "container" mechanism whereby a copy of the file is made in a container on the grid, or a "share" mechanism whereby the data continues to reside on the original machine, but can be accessed from the grid. Needless to say, these three inclusion mechanisms are completely orthogonal to the three access mechanisms discussed earlier.

Copy Inclusion

A common way of including data into a Legion Data Grid is by copying it into the grid with the *legion_cp* command. This command creates a Grid object or service that enables access to the data stored in a copy of the original file. The copy of the data may reside anywhere in the grid, and may also migrate throughout the Grid.

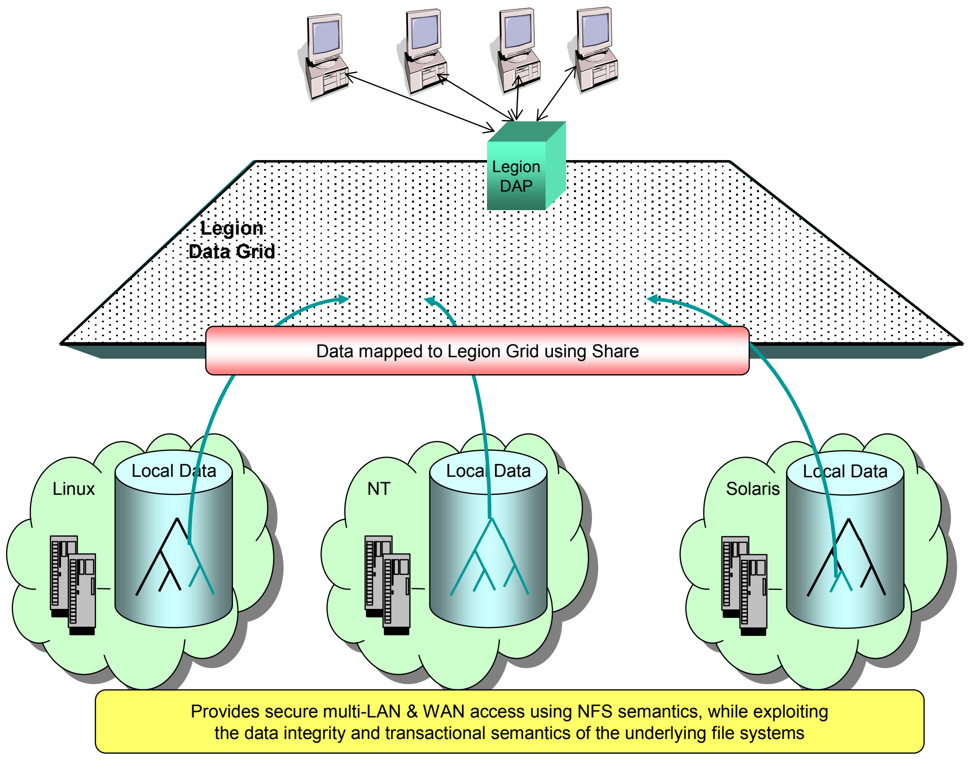
Container Inclusion

Data may be copied into a Grid container service as well. With this mechanism the contents of the original file are copied into a container object or service that enables access. The container mechanism reduces the overhead associated with having one service per file. Once again, data may migrate throughout the Grid.

Share Inclusion

The primary means of including data into a Legion Data Grid is with the *legion_export_dir* command. This command starts a daemon that maps a file or rooted directory in Unix or Windows NT into the data Grid. For example, *legion_export_dir C:\data /home/grimshaw/share-data* maps the directory *C:\data* on a Windows machine into the data Grid at */home/grimshaw/share-data*. Subsequently, files and subdirectories in *C:\data* can be accessed directly in a peer-to-peer fashion from anywhere else in the data Grid, subject to access control, without going through any sort of central repository. A Legion share is independent of the implementation of the underlying file system, whether a direct-attached disk on Unix or NT, an NFS-mounted file system, or some other file system such as a hierarchical storage management system.

Combining shares with DAPs effectively federates multiple directory structures into an overall file system, as shown in the figure below. Note that there may be as many DAPs as needed for scalability reasons.


Legion DAP
Legion Data Grid
Data mapped to Legion Grid using Share
Linux
Local Data
NT
Local Data
Solaris
Local Data
Provides secure multi-LAN & WAN access using NFS semantics, while exploiting the data integrity and transactional semantics of the underlying file systems

4.3 Distributed Processing

Research, engineering and product development depend on intensive data analysis and large simulations. In these environments, much of the work still requires computation-intensive data analysis – executing specific applications (which may be complex) with specific input data files (which may be very large or numerous) to create result files (which may also be large or numerous). For successful job execution, the data and application must be available, sufficient processing power and disk storage must be available, and the application's requirements for a specific operating environment must be met. In a typical network environment, the user must know where the file is, where the application is, and whether the resources are sufficient to complete the work. Sometimes, in order to achieve acceptable performance, the user or administrator must move data files or applications to the same physical location.

With Legion, users do not need to be concerned with these issues in most cases. Users have a single point of access to an entire Grid. Users log in, define application parameters and submit a program to run on available resources, which may be spread across distributed sites and multiple organizations. Input data is read securely from distributed sources without necessarily being copied to a local disk. Once an application is complete, computational resources are cleared of application remnants and output is written to the physical storage resources available in the Grid. Legion's distributed processing support includes several features, listed below.

4.3.1 Automated Resource Matching and File Staging

A Legion Grid user executes an application, referencing the file and application by name. In order to ensure secure access and implement necessary administrative controls, predefined policies govern where applications may be executed or which applications can be run on which data files. Avaki matches applications with queues and computing resources in different ways:

- **Through access controls.** For example, a user or application may or may not have access to a specific queue or a specific host computer.
- **Through matching of application requirements and host characteristics.** For example, an application may need to be run on a specific operating system, or require a particular library to be installed, or require a particular amount of memory.
- **Through prioritization.** For example, based on policies and load conditions.

Legion performs the routine tasks needed to execute the application. For example, Legion will move (or *stage*) data files, move application binaries and find processing power as needed, as long as the resources have been included into the Grid and the policies allow them to be used. If a data file or application must be migrated in order to execute the job, Legion does so automatically; the user does not need to move the files or know where the job was executed. Users need not worry about finding a machine for the application to run on, finding available disk space, copying the files to the machine and collecting results when the job is done.

4.3.2 Support for Legacy Applications – No Modification Necessary

Applications that use a Legion Grid can be written in any language, do not need to use a specific API, and can be run on the Grid without source code modification or recompilation. Applications can run anywhere at all on the Grid without regard to location or platform as long as resources are available that match the application's needs. This is critical in the commercial world where the sources for many third party applications are simply not available.

4.3.3 Batch Processing – Queues and Scheduling

With Legion, users can execute applications interactively or submit them to a queue. With queues, a Grid can group resources in order to take advantage of shared processing power, sequence jobs based on business priority when there is not enough processing power to run them immediately, and distribute jobs to available resources. Queues also permit allocation of resources to groups of users. Queues help insulate users from having to know where their jobs physically run. Users can check the status of a given job from anywhere on the Grid without having to know where it was actually processed. Administrator tasks are also simplified because a Legion Grid can be managed as a single system. Administrators can:

- Monitor usage from anywhere on the network.

- Preempt jobs, re-prioritize and re-queue jobs, take resources off the Grid for maintenance, or add resources to the Grid – all without interrupting work in progress.
- Establish policies based on time windows, load conditions or job limits. Policies based on time windows can make some hosts available for processing to certain groups only outside business hours or outside peak load times. Policies based on the number of jobs currently running or on the current processing load can affect load balancing by offloading jobs from overburdened machines.

4.3.4 Unifying Processing Across Multiple Heterogeneous Networks

A Legion Grid can be created from individual computers and local area networks or from clusters enabled by software such as the LSF™ or SGE™. If one or more queuing systems, load management systems or scheduling systems are already in place, Legion can interoperate with them to create virtual queues, thereby allowing resources to be shared across the clusters. This approach permits unified access to disparate, heterogeneous clusters, yielding a Grid that enables an increased level of sharing and allows computation to scale as required to support growing demands.

4.4 Security

Security was designed in the Legion architecture and implementation from the very beginning [4]. Legion's robust security is the result of several separate capabilities, such as authentication, authorization and data integrity, that work together to implement and enforce security policy. For Grid resources, the goal of Legion's security approach is to eliminate the need for any other software-based security controls, substantially reducing the overhead of sharing resources. With Legion security in place, users need only know their sign-on protocol and the Grid pathname where their files are located. Accesses to all resources are mediated by access controls set on the resources. We'll describe the security implementation in more detail in 5.3. Further details of the Legion security model are available in the literature [5][17].

4.5 Automatic Failure Detection and Recovery

Service downtimes and routine maintenance are common in a large system. During such times, resources may be unavailable. A Legion Grid is robust and fault-tolerant. If a computer goes down, Legion can migrate applications to other computers based on predefined deployment policies as long as resources are available that match application requirements. In most cases migration is automatic and unobtrusive. Since Legion is capable of taking advantage of all the resources on the Grid, it helps organizations optimize system utilization, reduce administrative overhead and fulfill service-level agreements.

- **Legion provides fast, transparent recovery from outages.** In the event of an outage, processing and data requests are rerouted to other locations, ensuring continuous operation. Hosts, jobs and queues automatically back up their current state, enabling them to restart with minimal loss of information if a power outage or other system interruption occurs.
- **Systems can be reconfigured dynamically.** If a computing resource must be taken offline for routine maintenance, processing continues using other resources. Resources can also be added to the Grid, or access changed, without interrupting operations.
- **Legion migrates jobs and files as needed.** If a job's execution host is unavailable or cannot be restarted, the job is automatically migrated to another host and restarted. If a host must be taken off the Grid for some reason, administrators can

migrate files to another host without affecting pathnames or users. Requests for that file are rerouted to the new location automatically.

5 The Legion Grid Architecture: Under the Covers

Legion is an object-based system comprised of independent objects, disjoint in logical address space, that communicate with one another by method invocation[2]. Method calls are non-blocking and may be accepted in any order by the called object. Each method has a signature that describes its parameters and return values (if any). The complete set of signatures for an object describes that object's interface, which is determined by its class. Legion class interfaces are described in an Interface Description Language (IDL), three of which are currently supported in Legion: the CORBA IDL [18], MPL [11] and BFS [6] (an object-based Fortran interface language). Communication is also supported for parallel applications using a Legion implementation of the MPI libraries. The Legion MPI supports cross-platform, cross-site MPI applications.

All things of interest to the system – for example, files, applications, application instances, users, groups – are objects. All Legion objects have a name, state (which may or may not persist), meta-data (<name, valueset> tuples) associated with their state and an interface[3]. At the heart of Legion is a three-level global naming scheme with security built into the core. The top-level names are human names. The most commonly used are *path names*, e.g., */home/grimshaw/my_sequences/seq_1*. Path names are all that most people ever see. The path names map to location independent identifiers called LOIDs. LOIDs are extensible self-describing data structures that include as a part of the name an RSA public key (the private key is part of the object's state). Consequently, objects may authenticate one another and communicate securely without the need for a trusted third party. Finally, LOIDs map to object addresses, which contain location-specific current addresses and communication protocols. An object address may change over time, allowing objects to migrate throughout the system, even while being used.

Each Legion object belongs to a class and each class is itself a Legion object. All objects export a common set of object-mandatory member functions (such as `deactivate()`, `ping()` and `getInterface()`), which are necessary to implementing core Legion services. Class objects export an additional set of class-mandatory member functions that enable them to manage their instances (such as `createInstance()` and `deleteInstance()`).

Much of the Legion object model's power comes from the role of Legion classes, since a sizeable amount of what is usually considered system-level responsibility is delegated to user-level class objects. For example, classes are responsible for creating and locating their instances and for selecting appropriate security and object placement policies. Legion core objects provide mechanisms that allow user-level classes to

[2] The fact that Legion is object-based does not preclude the use of non-object-oriented languages or non-object-oriented implementations of objects. In fact, Legion supports objects written in traditional procedural languages such as C and Fortran as well as object-oriented languages such as C++, Java, and Mentat Programming Language (MPL, a C++ dialect with extensions to support parallel and distributed computing).

[3] Note the similarity to the OGSA model, in which everything is a resource with a name, state and an interface.

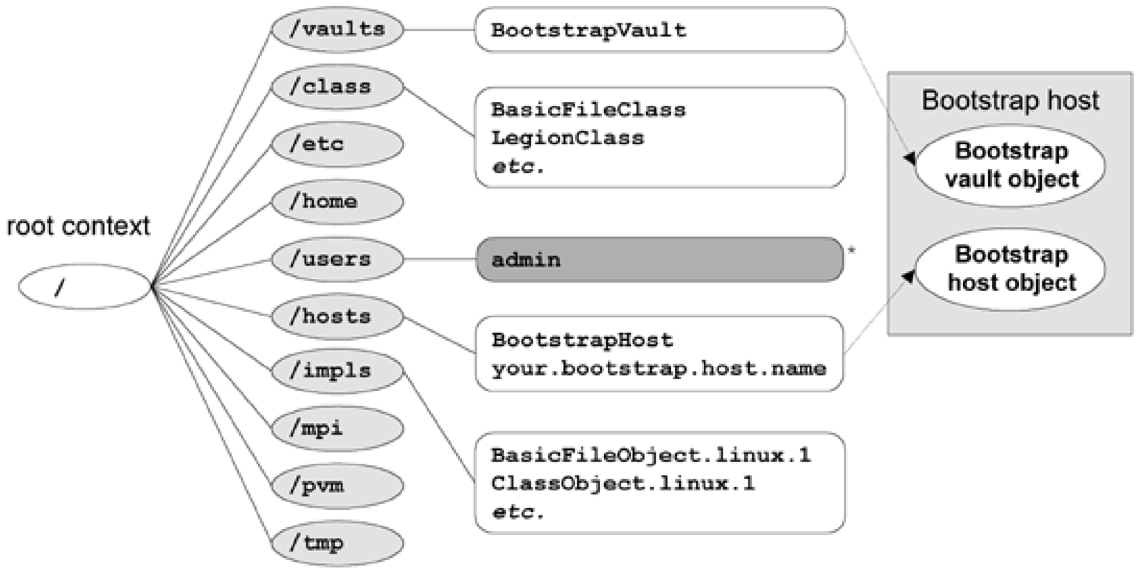
[7] There is a long history of languages for which this is not true. They are all interpreted languages in one form or another, either directly as in Perl, Python and shell scripts, or compiled into some form of byte code which is then executed by a virtual machine or interpreter, e.g., UCSD Pascal and Java.

implement chosen policies and algorithms. We also encapsulate system-level policy in extensible, replaceable class objects, supported by a set of primitive operations exported by the Legion core objects. This effectively eliminates the danger of imposing inappropriate policy decisions and opens up a much wider range of possibilities for the application developer.

5.1 Naming with Context Paths, LOIDs and Object Addresses

As in many distributed systems over the years, access, location, migration, failure and replication transparencies are implemented in Legion's naming and binding scheme. Legion has a three-level naming scheme. The top level consists of human-readable name spaces. There are currently two, context space and an attribute space. A context is an directory-like object that provides mappings from a user-chosen string to a Legion object identifier, called a *LOID*. A LOID can refer to a context object, file object, host object or any other type of Legion object. For direct object-to-object communication, a LOID must be bound to its low-level object address (*OA*), which is meaningful within the transport protocol used for communication. The process by which LOIDs are mapped to OAs is called the Legion *binding process*.

Contexts: Contexts are organized into a classic directory structure called *context space*. Contexts support operations that lookup a single string, return all mappings, add a mapping and delete a mapping. A typical context space has a well-known root context which in turn "points" to other contexts, forming a directed graph (Figure 2). The graph can have cycles, though it is not recommended.



We have built a set of tools for working in and manipulating context space, for example, *legion_ls*, *legion_rm*, *legion_mkdir*, *legion_cat* and many others. These are modeled on corresponding Unix tools. In addition there are libraries that mimic the *stdio* libraries for doing all of the same sort of operations on contexts and files (e.g., `creat()`, `unlink()`, etc.).

Legion Object IDentifiers: Every Legion object is assigned a unique and immutable LOID upon creation. The LOID identifies an object to various services (e.g., method invocation). The basic LOID data structure consists of a sequence of variable length binary string fields. Currently, four of these fields are reserved by the system. The first three play an important role in the LOID-to-object address binding mechanism: the first field is the *domain identifier*, used in the dynamic connection of separate Legion systems; the second is the *class identifier*, a bit string uniquely identifying the object's class within its domain; the third is an *instance number* that distinguishes the object from other instances of its class. LOIDs with an instance number field of length zero are defined to refer to class objects. The fourth and final field is reserved for security purposes. Specifically, this field contains an RSA public key for authentication and encrypted communication with the named object. New LOID types can be constructed to contain additional security information, location hints and other information in the additional available fields.

Object Addresses: Legion uses standard network protocols and communication facilities of host operating systems to support inter-object communication. To perform such communication Legion converts location-independent LOIDs into location-dependent communication system-level OAs through the Legion binding process. An OA consists of a list of *object address elements* and an *address semantic* field, which describes how to use the list. An OA element contains two parts, a 32-bit *address type* field indicating the type of address contained in the OA and the address itself, whose size and format depend on the type. The address semantic field is intended to express various forms of multicast and replicated communication. Our current implementation defines two OA types, the first consisting of a single OA element containing a 32-bit IP address, 16-bit port number and 32-bit unique id (to distinguish between multiple sessions that reuse a single IP/port pair). This OA is used by our UDP-based data delivery layer. The other is similar and uses TCP. Associations between LOIDs and OAs are called *bindings*, and are implemented as three-tuples. A binding consists of a LOID, an OA and a field that specifies the time at which the binding becomes invalid (including never). Bindings are first-class entities that can be passed around the system and cached within objects.

5.2 Metadata

Legion *attributes* provide a general mechanism to allow objects to describe themselves to the rest of the system in the form of metadata. An attribute is an *n*-tuple containing a *tag* and a list of *values*; the tag is a character string, and the values contain data that varies by tag. Attributes are stored as part of the state of the object they describe, and can be dynamically retrieved or modified by invoking object-mandatory functions. In general, programmers can define an arbitrary set of attribute tags for their objects, although certain types of objects are expected to support certain standard sets of attributes. For example, host objects are expected to maintain attributes describing the architecture, configuration and state of the machine(s) they represent.

Attributes are collected together into meta-data databases called *collections*. A collection is an object that "collects" attributes from a set of objects and can be queried to find all of the objects that have meet some specification. Collections can be organized into directed acyclic graphs, e.g., trees, in which collection data "flows" up and is aggregated in large-scale collections. Collections can be used extensively for resource discovery in many forms, e.g., for scheduling *("retrieve hosts where the one-minute load is less than some threshold")*.

5.3 Security

Security is an important requirement for grid infrastructures as well as for clients using grids. The definitions of security can vary depending on the resource accessed and the level of security desired. In general, Legion provides mechanisms for security while letting grid administrators configure policies that use these mechanisms. Legion provides mechanisms for authentication, authorization and data integrity/confidentiality.

The mechanisms are flexible enough to accommodate most policies desired by administrators. Legion explicitly accepts and supports that grids may comprise multiple organizations that have different policies and that wish to maintain independent control of all of their resources rather than trust a single global administrator.

In addition, Legion operates with existing network security infrastructures such as Network Address Translators (NATs) and firewalls. For firewall software, Legion requires port-forwarding to be set up. After this set-up, all Legion traffic is sent through a single bi-directional UDP port opened in the firewall. The UDP traffic sent through the firewall is transmitted using one of the data protection modes described in Section 5.3.3. In addition, Legion provides a sandboxing mechanism whereby different Legion users are mapped to different local Unix/Windows users on a machine and thus isolated from each other as well as the grid system on that machine. The sandboxing solution also permits the use of "generic" IDs so that Legion users who do not have accounts on that machine can continue to access resources on that machine using generic accounts.

Recall that LOIDs contain an RSA public key as part of their structure. Thus, naming and identity have been combined in Legion. Since public keys are part of object names, any two objects in Legion can communicate securely and authenticate without the need for a trusted third party. Further, any object can generate and sign a credential. Since all grid components, e.g., files, users, directories, applications, machines, schedulers, etc., are objects, security policies can be based not just on which person, but what component, is requesting an action.

5.3.1 Authentication

Users authenticate themselves to a Legion grid with the login paradigm. Logging in requires specifying a user name and password. The user name corresponds to an authentication object that is the user's proxy to the grid. The authentication object holds the user's private key, her encrypted password and user profile information as part of its persistent state. The password supplied during login is compared to the password in the state of the authentication object in order to permit or deny subsequent access to the grid. The state of the authentication object is stored on disk and protected by the local operating system. In other words, Legion depends on the local operating system to protect identities. Consequently, it requires careful configuration to place authentication objects on reliable machines. In a typical configuration, authentication objects for each administrative domain are restricted to a small number of trusted hosts.

A Legion system is neutral to the authentication mechanism. Past implementations have used the possession of a valid Kerberos credential as evidence of identity (as opposed to the password model above). In principle, any authentication mechanism, for example, NIS, LDAP, Kerberos, RSA SecurID, etc., can be used with Legion. Upon authentication, the authentication object generates and signs an encrypted (non-X.509) credential that is passed back to the caller. The credential is stored either in a temporary directory similar to the approach of Kerberos or stored as part of the internal state of the DAP. In either case, the credential is protected by the security of the underlying operating system.

Although login is the most commonly used method for users to authenticate themselves to a grid, it is not the only method. A Legion Data Access (DAP) enables accessing a data grid using the NFS protocol. The DAP maintains a mapping form the local Unix ID to the LOID of an authentication object. The DAP communicates directly with authentication objects to obtain credentials for Unix users. Subsequently, the DAP uses the credentials to access data on the user's behalf. With this access method, users may be unaware that they are using Legion's security system to access remote data. Moreover, clients of the DAP typically do not require installing Legion software.

Credentials are used for subsequent access to grid resources. When a user initiates an action, her credentials are securely propagated along the chain of resources accessed in the course of the action using a mechanism called “implicit parameters”. Implicit parameters are the grid equivalent of environment variables – they are tuples of the form <name, value> and are used to convey arbitrary information to objects.

5.3.2 Authorization

Authorization addresses the issue of who or what can do which operations on what objects. Legion is policy-neutral with respect to authorization, i.e., a large range of policies are possible. Legion/Avaki ships with an access control list (ACL) policy, which is a familiar mechanism for implementing authorization. For each action (or function) on an object, e.g., a “read” or a “write” on a file object, there exists an “allow” list and a “deny” list. Each list can contain the names of other objects, typically, but not restricted to authentication objects such as “*/users/jose*”. A list may also contain a directory, which in turn contains a list of objects. If the directory contains authentication objects, then it functions as a group. Selecting which authentication objects are listed in directories can form arbitrary groups. The allow list for an action on an object thus identifies a set of objects (often, authentication objects, which are proxies for humans) that are allowed to perform the action. Likewise, the deny list identifies a set of objects which are not allowed to perform the action. If an object is present in the allow list as well as the deny list, we choose the more restrictive option, i.e., the object is denied permission. An access control list can be manipulated in several ways; one way provided in Legion is with a “chmod” command that maps the intuitive “+/-” and “rwx” options to the appropriate allow and deny lists for sets of actions.

Access control lists can be manipulated to permit or disallow arbitrary sharing policies. They can be used to mimic the usual read-write-execute semantics of Unix, but can be extended to include a wider variety of objects than just files and directories. Complex authorizations are possible. For example, if three mutually-distrustful companies wish to collaborate in manner wherein one provides the application, another provides data and the third provides processing power, they can do so by setting the appropriate ACLs in a manner that permits exactly the actions desired and none other. In addition, the flexibility of creating groups at will is a benefit to collaboration. Users do not have to submit requests to create groups to system administrators – they can create their own groups by linking in existing users into a directory and they can set permissions for these groups on their own objects.

5.3.3 Data Integrity and Data Confidentiality

Data integrity is the assurance that data has not been changed, either maliciously or inadvertently, either in storage or in transmission, without proper authorization. Data confidentiality (or privacy) refers to the inability of an unauthorized party to view/read/understand a particular piece of data, again either in transit or in storage. For on-the-wire protection, i.e., protection during transmission, Legion follows the lead of almost all contemporary security systems in using cryptography. Legion transmits data in one of three modes: *private*, *protected* and *none*. Legion uses the OpenSSL libraries for encryption. In the private mode, all data is fully encrypted. In the protected mode, the sender computes a checksum (e.g., MD5) of the actual message. In the third mode, “none”, all data is transmitted in the clear except for credentials. This mode is useful for applications that do not want to pay the performance penalty of encryption and.or checksum calculations. The desired integrity mode is carried in an implicit parameter that is propagated down the call chain. If a user sets the mode to *private* then all subsequent communication made by the user or on behalf of the user will be fully encrypted. Since different users may have different modes of encryption, different communications with the same object may have different encryption modes. Objects in the call chain, for example, the DAP, may override the mode if deemed appropriate.

For on-the-host data protection, i.e., integrity/confidentiality during storage on a machine, Legion relies largely on the underlying operating system. Persistent data is stored on local disks with the appropriate permissions such that only appropriate users can access them.

6 Core Legion Objects

In this section we delve deeper into the Legion implementation of a Grid. In particular, we discuss some of the core objects that comprise a Grid. The intention of this section is to emphasize how a clean architectural design encourages the creation of varied services in a grid. In the case of Legion, these services are implemented naturally as objects. We do not intend this section to be a catalogue of all Legion objects for two reasons: one, space considerations prevent an extensive catalogue, and two, an abiding design principle in Legion is extensibility, which in turn means that a complete catalogue is intentionally impossible. Also, in this section, we stop at the object level. All objects in turn use a common communication substrate in order to perform their tasks. This communication is based on dataflow graphs, with each graph encapsulating a complex, reflection-based, remote method invocation. In turn, the method invocation involves a flow-controlled message-passing protocol. Each of these topics is interesting in itself; however, we refer the reader to literature on Legion as well as other systems regarding these topics [17][24].

6.1 Class Objects

Class objects manage particular instances. For example, a class object may manage all running instances of an application. Managing instances involves creating them on demand, destroying them when required, managing their state and keeping track of them.

Every Legion object is defined and managed by its class object. Class objects are *managers* and *policy makers* and have system-like responsibility for creating new instances, activating and deactivating them, and providing bindings for clients. Legion encourages users to define and build their own class objects. These two features – class object management of the class's instances and applications programmers' ability to construct new classes – provide flexibility in determining how an application behaves and further support the Legion philosophy of enabling flexibility in the kind and level of functionality.

Class objects are ideal for exploiting the special characteristics of their instances. We expect that a vast majority of programmers will be served adequately by existing *metaclasses*. *Metaclass objects* are class objects whose instances are themselves class objects. Just as a normal class object maintains implementation objects for its instances, so too does a metaclass object. A metaclass object's implementation objects are built to export the class-mandatory interface and to exhibit a particular class functionality behind that interface. To use one, a programmer simply requests a create on the appropriate metaclass object, and configures the resulting class object with implementation objects for the application in question. The new class object then supports the creation, migration, activation and location of these application objects in the manner defined by its metaclass object.

For example, consider an application that requires a user to have a valid software license in order to create a new object, e.g., a video-on-demand application in which a new video server object is created for each request. To support this application, the developer could create a new metaclass object for its video server classes, the implementation of which would add a license check to the object creation method.

6.2 Hosts

Legion host objects abstract processing resources in Legion. They may represent a single processor, a multiprocessor, a Sparc, a Cray T90 or even an aggregate of multiple hosts. A host object is a machine's representative to Legion: it is responsible for executing objects on the machine, protecting the machine from Legion users, protecting user objects from each other, reaping objects and reporting object exceptions. A host object is also ultimately responsible for deciding which objects can run on the machine it represents. Thus, host objects are important points of security policy encapsulation.

Aside from implementing the host-mandatory interface, host object implementations can be built to adapt to different execution environments and suit different site policies and underlying resource management interfaces. For example, the host object implementation for an interactive workstation uses different process creation mechanisms than implementations for parallel computers managed by batch queuing systems. In fact, the standard host object has been extended to submit jobs to a queuing system. Another extension to the host object has been to create container objects which permit execution of only one type of object on a machine. An implementation to address the performance problems might use threads instead of processes. This design improves the performance of object activation, and also reduces the cost of method invocation between objects on the same host by allowing shared address space communication.

Whereas host objects present a uniform interface to different resource environments, they also (and more importantly) provide a means for resource providers to enforce security and resource management policies within a Legion system. For example, a host object implementation can be customized to allow only a restricted set of users access to a resource. Alternatively, host objects can restrict access based on code characteristics (e.g., accepting only object implementations that contain proof-carrying code demonstrating certain security properties, or rejecting implementations containing certain "restricted" system calls).

We have implemented a spectrum of host object choices that trade-off risk, system security, performance and application security. An important aspect of Legion site autonomy is the freedom of each site to select the existing host object implementation that best suits their needs, extend one of the existing implementations to suit local requirements, or to implement a new host object starting from the abstract interface. In selecting and configuring host objects, a site can control the use of their resources by Legion objects.

6.3 Vaults

Vault objects are responsible for managing other Legion objects' persistent representations (OPRs). Much in the same way that hosts manage active objects' direct access to processors, vaults manage inert objects on persistent storage. A vault has direct access to a storage device (or devices) on which the OPRs it manages are stored. A vault's managed storage may include a portion of a Unix file system, a set of databases or a hierarchical storage management system. The vault supports the creation of OPRs for new objects, controls access to existing OPRs and supports the migration of OPRs from one storage device to another. Manager objects manage the assignment of vaults to instances: when an object is created, its vault is chosen by the object's manager. The selected vault creates a new, empty OPR for the object and supplies the object with its state. When an object migrates, its manager selects a new target vault for its OPR.

6.4 Implementation Objects

Implementation objects encapsulate Legion object executables. The executable itself is treated much like a Unix file (i.e., as an array of bytes) so the implementation object interface naturally is similar to a Unix file interface. Implementation objects are also write-once, read-many objects – no updates are permitted after the executable is initially stored.

Implementation objects typically contain executable code for a single platform, but may in general contain any information necessary to instantiate an object on a particular host. For example, implementations might contain Java byte code, Perl scripts or high-level source code that requires compilation by a host. Like all other Legion objects, implementation objects describe themselves by maintaining a set of attributes. In their attributes, implementation objects specify their execution requirements and characteristics which may then be exploited during the scheduling process [3]. For example, an implementation object may record the type of executable it contains, its minimum target machine requirements, performance characteristics of the code, etc.

Class objects maintain a complete list of (possibly very different) acceptable implementation objects appropriate for their instances. When the class (or its scheduling agent) selects a host and implementation for object activation, it selects them based on the attributes of the host, the instance to be activated and the implementation object.

Implementation objects allow classes a large degree of flexibility in customizing the behavior of individual instances. For example, a class might maintain implementations with different time/space trade-offs and select between them depending on the currently available resources. To provide users with the ability to select their cost/performance trade-offs, a class might maintain both a slower, low-cost implementation and faster, higher-cost implementation. This approach is similar to abstract and concrete types in Emerald.

6.5 Implementation Caches

Implementation caches avoid storage and communication costs by storing implementations for later reuse. If multiple host objects share access to some common storage device they may share a single cache to further reduce copying and storage costs. Host objects invoke methods on the implementation caches in order to download implementations. The cache object either finds it already has a cached copy of the implementation or it downloads and caches a new copy. In either case, the cache object returns the executable's path to the host. In terms of performance, using a cached binary results in object activation being only slightly more expensive than running a program from a local file system.

Our implementation model makes the invalidation of cached binaries a trivial problem. Since class objects specify the LOID of the implementation to use on each activation request, a class need only change its list of binaries to replace the old implementation LOID with the new one. The new version will be specified with future activation requests, and the old implementation will simply no longer be used and will time-out and be discarded from caches.

7 The Transformation From Legion to Avaki

Legion began in late 1993 with the observation that dramatic changes in wide-area network bandwidth were on the horizon. In addition to the expected vast increases in bandwidth, other changes such as faster processors, more available memory, more disk space, etc. were expected to follow in the usual way as predicted by *Moore's Law*. Given the dramatic changes in bandwidth expected, the natural question was, how will this bandwidth be used? Since not just bandwidth will change, we generalized the question to, "Given the expected changes in the physical infrastructure – what sorts of applications will people want, and given that, what is the system software infrastructure that will be needed to support those applications?" The Legion project was born with the determination to build, test, deploy and ultimately transfer to industry, a robust, scalable, Grid computing software infrastructure. We followed the classic design paradigm of first determining requirements, then completely designing the system architecture on paper after numerous design meetings, and finally, after a year of design work, coding. We made a decision to write from scratch rather than extend and modify our existing system (Mentat [11]) that we had been using as a prototype. We felt that only by starting from scratch could we ensure adherence to our architectural principles. First funding was obtained in early 1996, and the first line of Legion code was written in June of 1996.

The basic architecture was driven by the principles and requirements described above, and by the rich literature in distributed systems. The resulting architecture was reflective, object-based to facilitate encapsulation, extensible, and was in essence an operating system for Grids. We felt strongly that having a common, accepted underlying architecture and set of services that can be assumed is critical for success in Grids. In this sense the Legion architecture anticipated the drive to Web Services and OGSI. Indeed, the Legion architecture is very similar to OGSI [8].

By November, 1997 we were ready for our first deployment. We deployed Legion at UVa, SDSC, NCSA and UC Berkeley for our first large scale test and demonstration at Supercomputing 1997. In the early months keeping the MTBF over twenty hours under continuous use was a challenge. This is when we learned several valuable lessons. For example, we learned that the world is not "fail-stop". While we intellectually knew this – it was really brought home by the unusual failure modes of the various hosts in the system.

By November 1998, we had solved the failure problems and our MTBF was in excess of one month, and heading towards three months. We again demonstrated Legion – now on what we called NPACI-Net. NPACI-Net consisted of hosts at UVa, Caltech, UC Berkeley, IU, NCSA, the University of Michigan, Georgia Tech, Tokyo Institute of Technology and the Vrije Universiteit, Amsterdam. By that time dozens of applications had been ported to Legion from areas as diverse as materials science, ocean modeling, sequence comparison, molecular modeling and astronomy. NPACI-Net continues today with additional sites such as the University of Minnesota, SUNY Binghamton and PSC. Supported platforms include Windows 2000, the Compaq iPaq, the T3E and T90, IBM SP-3, Solaris, Irix, HPUX, Linux, True 64 Unix and others.

From the beginning of the project a "technology transfer" phase had been envisioned in which the technology would be moved from academia to industry. We felt strongly that Grid software would move into mainstream business computing only with commercially-supported software, help lines, customer support, services and deployment teams. In 1999, Applied MetaComputing was founded to carry out the technology transition. In 2001, Applied MetaComputing raised $16M in venture capital and changed its name to AVAKI. The company acquired legal rights to Legion from the University of Virginia and renamed Legion to "Avaki". Avaki was released commercially in September, 2001. Avaki is an extremely hardened, trimmed-down, focused-on-commercial-requirements version of Legion. While the name has changed, the core architecture and the principles on which it operates remain the same.

7.1 Avaki Today

The first question must answer before looking at Avaki is “Why commercialize Grids?” Many of the technological challenges faced by companies today can be viewed as variants of the requirements of Grid infrastructures. Today, science and commerce is increasingly a global enterprise, in which people collaborating on a single research project, engineering project or product development effort may be located over distances spanning a country or the world. The components of the project or product – data, applications, processing power and users – may be in distinct locations from one another. This scenario is particularly true in the life sciences, in which there are large amounts of data generated by many different organizations, both public and private, around the world. The attractive model of accessing all resources as if they were local runs into several immediate challenges. Administrative controls set up by organizations to prevent unauthorized accesses to resources hinder authorized accesses as well. Differences in platforms, operating systems, tools, mechanisms for running jobs, data organizations, etc. impose a heavy cognitive burden on users. Changes in resource usage policies and security policies affect the day-to-day actions of users. Finally, large distances act as barriers to the quick communication necessary for collaboration. Consequently, users spend too much time on the procedures for accessing a resource and too little time using the resource itself. These challenges lower productivity and hinder collaboration.

Grids offer the promise to solve the challenges facing collaboration by providing the mechanisms for easy and secure access to resources. Academic and government-sponsored Grid infrastructures, such as Legion, have been used to construct long-running Grids accessing distributed, heterogeneous and potentially faulty resources in a secure manner. There are clear benefits in making Grids available to an audience wider than academia or government. However, clients from industry make several demands that are not traditionally addressed by academic projects. For example, industry clients typically demand a product that is supported by a company whose existence can be assured for a reasonably long period. Moreover, the product must be supported by a professional services team that understands how to deploy and configure the product. Clients demand training engagements, extensive documentation, always-available support staff and a product roadmap that includes their suggestions regarding the product. Such clients do not necessarily view open source or free software as requisites; they are willing to pay for products and services that will improve the productivity of their users.

A successful technology is one that can transition smoothly from the comfort and confines of academia to the demanding commercial environment. Several academic projects are testament to the benefits of such transitions; often, the transition benefits not just the user community but the quality of the product as well. We believe Grid infrastructures are ready to make such a transition. Legion had been tested in non-industry environments from 1997 onwards during which time we had the opportunity to test the basic model, scalability, security features, tools and development environment rigorously. Further improvement required input from a more demanding community with a vested interest in using the technology for their own benefit. The decision to commercialize Grids, in the form of the Avaki 2.x product and beyond was inevitable.

7.2 How are Grid requirements relevant to a commercial product?

Despite changes to the technology enforced by the push to commercialization, the basic technology in Avaki 2.x remains the same as Legion. All of the principles and architectural features discussed earlier continue to form the basis of the commercial product. As a result, the commercial product continues to meet the requirements outlined in the introduction. These requirements follow naturally from the challenges faced by commercial clients who attempt to access distributed, heterogeneous resources in a secure manner.

Consider an everyday scenario in a typical life sciences company as a use case. Such a company may have been formed by the merger of two separate companies, with offices located in different cities. The original companies would have purchased their IT infrastructure, i.e., their machines, their operating systems, their file systems and their tools independent of each another. The sets of users at the two companies would overlap to a very small extent, often not at all. After the merger, the companies may be connected by VPN (virtual private network) software, but their traffic goes over the public network. Users would expect to use the resources of the merged company shortly after the merger. Moreover, they may expect to use public databases as well as databases managed by other partner companies.

The above scenario is common in several companies not only in life sciences but also other verticals, such as finance, engineering design and natural sciences. Any solution that attempts to solve this scenario must satisfy several, if not all of the Grid requirements. For example, the different IT infrastructures lead to the requirement of heterogeneity. The desire to access resources located remotely and with different access policies leads to complexity. Since resources are accessed over a public network, fault-tolerance will be expected. Besides, as resources are increased, the mean time to failure of the infrastructure as a whole increases surprisingly rapidly even if the individual resources are highly reliable. Users in the merged company will expect the merged IT infrastructure to be secure from external threats, but in addition, they may desire security from each other, at least until enough trust has been built between users of the erstwhile companies. Accessing remote computing resources leads to binary management and support for applications written in any language. The inclusion of private, public and partnered databases in accesses demands a scalable system that can grow as the numbers and sizes of the databases increase. Managers of these databases will expect to continue controlling the resources they make available to others, as will managers of other resources, such as system administrators. Users will expect to access resources in as simple a manner as possible – specifying just a name is ideal, but specifying any of location, method, behavior, etc. is too cumbersome. Finally, users will expect to reuse these solutions. In other words, the infrastructure must continue to exist beyond one-time sessions. Any software solution that addresses all of these requirements is a Grid infrastructure.

7.3 What is retained, removed, reinforced?

Avaki 2.x retains the vision, the philosophy, the principles and the underlying architecture of Legion. It eliminates some of the more esoteric features and functions present in Legion. It reinforces the robustness of the infrastructure by adding more stringent error-checking and recommending safer configurations. Moreover, it increases the usability of the product by providing extensive documentation, configuration guidelines and additional tools.

As a company, AVAKI necessarily balanced retaining only those features and functions of the product that addressed immediate revenue against retaining all possible features in Legion. For example, with Legion, we created a new kind of file object called a 2D file partly to support applications that access large matrices and partly to demonstrate the ability to create custom services in Legion. Since a demand for 2D files was low in the commercial environment, Avaki 2.x does not support this object, although the code base includes it. Likewise, until clients request Avaki's heterogeneous MPI tools, support for this feature is limited.

The Legion vision of a single-system view of Grid components is a compelling one, and we expect it to pervade future releases of Avaki and products from AVAKI. In addition, we expect several of the philosophical aspects, such as the object model, naming and well-designed security, will continue. Underlying principles such as "no privileged user requirement" have been popular with users, and we expect to continue them. The underlying architecture may change, especially with the influence of web services or other emerging technologies. Likewise, the implementation of the product may change, but that change is an irrelevant detail to Grid customers.

The commercial product reinforces many of the strengths of Legion. For example, with the commercial product, installation and Grid creation is easier than before – these actions require a total of two commands. Several commands have been simplified while retaining their Unix-like syntax. Configuring the Grid in the myriad ways possible has been simplified. The underlying communication protocol as well as several tools and services have been made more robust. Error messaging has been improved, and services have been made more reliable. Support for several of the strengths of Legion – transparent data access, legacy application management, queuing, interfaces to 3rd-party queuing and MPI systems, parameter-space studies – have been improved.

8 Meeting the Grid Requirements with Legion

Legion continues to meet the technical grid requirements outlined in Section 2. In addition, it meets commercial requirements for grids as well. In this section we discuss how Legion and Avaki meet the technical grid requirements by revisiting each requirement identified in Section 2:

- ***Security***. As described in 5.3 and in the literature [4][5], security is a core aspect of both the Legion architecture and the Legion implementation. Legion addresses authentication, authorization, data integrity and firewalls in a flexible manner that separates policy and mechanism cleanly.
- ***Global name space***: A global namespace is fundamental to our model and is realized by context space and LOIDs.
- ***Fault-tolerance***. Fault-tolerance remains one of the most significant challenges in Grid systems. We began with a belief that there is no good one-size-fits-all solution to the fault-tolerance problem in Grids [13] [23]. Legion addresses fault-tolerance for limited classes of applications. Specifically stateless application components, jobs in high-throughput computing, simple K-copy objects that are resilient to K-1 host failures and MPI applications that have been modified to use Legion save-and-restore state functions. There remains however, a great deal of work to be done in the area of fault-tolerance for Grids. Indeed this is the thrust of the IBM efforts in autonomic computing.
- ***Accommodating heterogeneity***. Operating system and architecture heterogeneity is actually one of the easier problems to solve, and a problem for which good solutions have been available for over two decades. In general there are two basic challenges to overcome: data format conversions between heterogeneous architectures (e.g., those that arise from a cross-architecture RPC or reading binary data that was written on another architecture) and executable format differences including different instruction sets (you can't use the same executable on all platforms[7]).

  Data format conversions have traditionally been done in one of two ways. In the first technique, exemplified by XDR, all data is converted from native format to a standard format when sent, and then converted back to the native format when received. The second technique has been called "receiver makes right". The data is sent in its native format, along with a metadata tag that indicates which format the data is in.

Upon receipt the receiver checks the tag. If the data is already of the native format no conversion is necessary. If the data is of a different format, then the data is converted on the fly. Legion uses the second technique. Typed data is “packed” into buffers. When data is “unpacked” we check the metadata tag associated with the buffer and convert the data if needed. We have defined packers and unpackers (including all pair-wise permutations) for Intel (32 bit), Sun, SGI (32 and 64 bit), HP, DEC Alpha (32 and 64 bit), IBM RS6000, and Cray T90 and C90.

- ***Binary management***. The Legion run-time ensures that the appropriate binaries, if they exist, are available on a host whenever needed. Each class object maintains a list of implementations available for the class and each implementation is itself a Legion object with methods to read the implementation data. At run-time the schedulers will ensure that only hosts for which binaries are available are selected. Once a host has been selected and an appropriate implementation chosen the class object asks the host to instantiate an object instance. The host asks the implementation cache for a path to the executable, and the cache down-loads the implementation if necessary. No user-level code is involved in this process.
- ***Multi-language support***. Avaki is completely agnostic to programming language. Applications have been written in C, C++, Fortran, SNOBOL, Perl, Java and shell–scripting languages.
- ***Scalability***. A truly scalable system can keep growing without performance degradation. In order to have a scalable system there must be no hotspots. System components should not become overloaded as the system scales to millions of hosts and billions of objects. This means that as the number of users and hosts increases the number of “servers” that provide basic services must increase as well without requiring a super-linear increase in the amount of traffic.

  Before we begin let’s examine where hot-spots would be most likely to occur. The first and most obvious place is the root of the class structure and hence the binding structure. A second point is near the top of the context space tree, at the root and perhaps the top level or two of directories. Once again these are globally-shared, and frequently-accessed data structures.

  We achieve scalability in Legion through three aspects of the architecture: hierarchical binding agents, distribution of naming and binding to multiple class objects and contexts, and replication and cloning of classes and objects.

  A single binding agent would be no more scalable than a single class object. Therefore we provide for a multitude of binding agents, typically one per host, that are arranged in a tree and behave much as a software combining tree in high-performance shared memory architectures. The effect is to reduce the maximum lookup rate.

  In the case of the LOID to OA mapping Legion detects stale bindings (allowing safe aggressive caching, discussed above) and by distributing the ultimate binding authority to class objects. There is different class object for every type in the system, e.g., for program1, for BasicFileObjects, etc.

  In order to prevent frequently used classes, e.g., BasicFileClass, from becoming hot-spots Legion supports *class cloning* and object replication. When a class is cloned a new class object is instantiated. The new class has a different LOID but all of the same implementations. When a new instance is created one of the clones is selected (typically round-robin) to perform the instantiation and to manage the instance in the future.

- ***Persistence***. Persistence in Legion is realized with Vaults (section 6.3), and as far as most users are concerned, with the Legion Data Grid described above.
- ***Extensibility***. No matter how carefully a system is designed and crafted it will not meet the needs of all users. The reason is that different users, applications and organizations have different requirements and needs, e.g., some applications require a coherent distributed file system while others have weaker semantic requirements. Some organizations require the use of Kerberos credentials while others do not. The bottom line is that a restrictive system will be unable to meet all current and future requirements.

  Legion was designed to be customized and tailored to different application or organizational requirements. These customizations can take place either in the context of a closed system, e.g., a total system change that affects all objects, or in the context of a local change that affects only a subset of objects and does not affect the interoperability of the modified objects and other objects in the system.

  Broadly, there are three places where customization can occur: modification of core daemon objects, definition of new metaclasses to change meta-behaviors for a class of objects (interobject protocols), and changes in the Legion run-time libraries to change how Legion is implemented inside an object (intraobject protocols).

  The core daemons, viz., hosts, vaults, classes, contexts and implementation objects and schedulers can be modified and extended by system administrators to over-ride the default behaviors.

  Legion class objects manage their instances and determine how their instances are instantiated. For example, the default, vanilla class object behavior instantiates its instances with one instance per address space and the class object gives out the actual object address to binding agents. Similarly, the state of the instances is stored in vaults, and there is one copy of the state. In Legion, the meta-object protocols and interactions can be selectively modified on a class-by-class basis. This is done by overriding class object behaviors with a new implementation. The result is a new meta-class object.

  The final way to modify and extend the system is by modifying the Legion Run-Time Libraries (LRTL) [27], or by changing the configuration of the event handlers. Legion is a reflective system, so the internal representation of Legion in objects is exposed to programmers. Modifications to that representation will change behavior. The LRTL was designed to be changed, and to have the event “stack” modified. To date we have used this primarily for modifications to the security layer and to build performance-optimized implementations for our MPI implementation.
- ***Site autonomy***. Site autonomy is critical for Grid systems. We believe that very few individuals, and no enterprises, will agree to participate in a large-scale Grid system if it requires giving up control of their resources and control of local policy decisions such as who can do what on their resources. Site autonomy in Legion is guaranteed by two mechanisms: protecting the physical resources and setting the security policies of the objects used by local users.
  Host and vault objects represent a site’s physical resources. They are protected by their respective access control lists. As discussed earlier the access control policy can be over-riden by defining new a new host or vault class. The result is a change in local policy. In general the hosts and vault policies can be set to whatever is needed by the enterprise.

- ***Complexity management***. Complexity is addressed in Legion by providing a complete set of high-level services and capabilities that mask the underlying infrastructure from the user so that, by default the user does not have to think. At level of programming the Grid, Legion takes a classic approach of using the object paradigm to encapsulate complexity in objects whose interfaces define behavior but not implementation.

9 Emerging Standards

Recently at the Global Grid Forum an Open Grid Services Architecture (OGSA) [8] was proposed by IBM and the Globus PIs. The proposed architecture has many similarities to the Avaki architecture. One example of the congruence is that all objects (called Grid Resources in OGSA) have a name, an interface, a way to discover the interface, meta-data and state, and are created by factories (analogous to Avaki class objects). The primary differences in the core architecture lie in the RPC model, the naming scheme and the security model.

The RPC model differences are of implementation – not of substance. This is a difference that Avaki intends to address by becoming Web Services compliant, i.e., by supporting XML/SOAP and WSDL).

As to the naming model, both offer two low-level name schemes in which there is an immutable, location-independent name (GSH in OGSA, LOID in Avaki) and a lower-level "address" (a WSDL GSR in OGSA and an OA in Avaki). The differences though are significant. First, OGSA names have no security information in them at all, requiring the use of alternative mechanism to bind name and identity. We believe this is critical as Grids become wide-spread and consist of thousands of diverse organizations. Second, binding resolvers in OGSA currently are location- and protocol-specific, severely reducing the flexibility of the name resolving process. To address these issues Avaki has proposed the Secure Grid Naming Protocol (SGNP) [1] to the GGF as an open standard for naming in Grids. SGNP fits quite well with OGSA, and we are actively working with IBM and others within the GGF working group process to find the best solution for naming.

It is interesting to note the similarity of Legion to OGSI. Both architectures have at their core the notion of named entities that interact using method calls. Both use multi-layer naming schemes and reflective interface discovery, and defer to user-defined objects how those names will be bound, providing for a wide variety of implementations and semantics. The similarity is not a surprise as both draw on the same distributed systems literature.

As to security, at this juncture there is no security model in OGSA. That is a shortcoming that we certainly expect to be remedied soon – at least as far as authentication and data integrity. The Globus group [7] has a long history in this area [2]. The Avaki security model was designed to be flexible, and has identity included in names. Further, the Avaki model has a notion of access control through replaceable modules called "MayI" that implement access control. The default policy is access control lists.

There are many non-architectural differences between Avaki and OGSA – Avaki is a complete, implemented system with a wealth of services. OGSA is a core architectural proposal – not a complete system. It is expect that higher level interfaces and components will be added to the basic OGSA substrate over time at GGF.

Finally, at Avaki we are fully committed to OGSI moving forward. We feel strongly that a foundation of good standards will accelerate the development of the Grid market and benefit everybody in the community, both users and producers of Grid software.

10 Summary

The Legion project was begun in late 1993 to construct and deploy large-scale metasystems for scientific computing, though with a design goal to be a general purpose meta-operating system. Since then "metacomputing" has become "Grid computing" and the whole concept of Grid computing has begun to move into the mainstream. From the beginning we argued that Grid systems should be designed and engineered like any other large scale software system – first by examining the requirements, then by designing an architecture to meet the requirements, and finally by building, deploying and testing the resulting software.

Thus Legion was born. Today, almost a decade later, not only have the basic design principles and architectural features stood the test of time, but we have a production operational Grid system that has demonstrated utility not only in academia, but in industry as well where robust software and hard return-on-investment are requirements.

In this paper we presented what we believe are some of the fundamental architectural requirements of Grids, as well as our design philosophy that we used in building Legion. We also presented a glimpse of how Legion is used in the field, as well as at some basic aspects of the Legion architecture. We concluded with a discussion of the transformation of Legion into Avaki, and how we see Avaki fitting into the context of OGSI. The presentation was, of necessity, rather brief. We encourage the interested reader to follow references embedded in the text to much more complete descriptions.

11 References

[1] J. Apgar, A.S. Grimshaw, S. Harris, M.A. Humphrey and A. Nguyen-Tuong, "Secure Grid Naming Protocol: Draft Specification for Review and Comment," http://sourceforge.net/projects/sgnp.

[2] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer and V. Welch, "A National-Scale Authentication Infrastructure," *IEEE Computer*, 33(12):60-66, 2000.

[3] S.J. Chapin, D. Katramatos, J.F. Karpovich and A.S. Grimshaw, "Resource Management in Legion," *Journal of Future Generation Computing Systems*, vol. 15, pp. 583-594, 1999.

[4] S.J. Chapin, C. Wang, W.A. Wulf, F.C. Knabe and A.S. Grimshaw, "A New Model of Security for Metasystems," *Journal of Future Generation Computing Systems*, vol. 15, pp. 713-722, 1999.

[5] A.J. Ferrari, F.C. Knabe, M.A. Humphrey, S.J. Chapin and A.S. Grimshaw, "A Flexible Security System for Metacomputing Environments," *7th International Conference on High-Performance Computing and Networking Europe (HPCN'99),* April 1999, Amsterdam: 370-380.

[6] A.J. Ferrari and A.S. Grimshaw, "Basic Fortran Support in Legion," Technical Report CS-98-11, Department of Computer Science, University of Virginia, March 1998.

[7] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *International Journal of Supercomputing Applications*, 1997.

[8] I. Foster, C. Kesselman, J. Nick and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," http://www.Gridforum.org/drafts/ogsi-wg/ogsa_draft2.9_2002-06-22.pdf

[9] A.S. Grimshaw, "Enterprise-Wide Computing," *Science*, 256: 892-894, Aug 12, 1994.

[10] A.S. Grimshaw and W.A. Wulf, "The Legion Vision of a Worldwide Virtual Computer," *Communications of the ACM*, 40(1): 39-45, Jan 1997.

[11] A.S. Grimshaw, J.B. Weissman and W.T. Strayer, "Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing,'' ACM *Transactions on Computer Systems*, 14(2): 139-170, May 1996.

[12] A. S. Grimshaw *et al.*, "Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems," Department of Computer Science Technical Report CS-98-12, University of Virginia, June 1998.

[13] A. Nguyen-Tuong and A.S. Grimshaw, “Using Reflection for Incorporating Fault-Tolerance Techniques into Distributed Applications,” *Parallel Processing Letters, vol. 9, No. 2 (1999)*, 291-301.

[14] A.S. Grimshaw, A.J. Ferrari, F.C. Knabe and M.A. Humphrey, “Wide-Area Computing: Resource Sharing on a Large Scale," *IEEE Computer*, 32(5): 29-37, May 1999.

[15] A.S. Grimshaw, A.J. Ferrari, G. Lindahl and K. Holcomb, “Metasystems," *Communications of the ACM*, 41(11): 486-55, Nov 1998.

[16] A.S. Grimshaw, A.J. Ferrari, F.C. Knabe and M.A. Humphrey, “Wide-Area Computing: Resource Sharing on a Large Scale," *IEEE Computer*, 32(5): 29-37, May 1999.

[17] A.S. Grimshaw, M.J. Lewis, A.J. Ferrari and J.F. Karpovich, "Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems," Technical Report CS-98-12, Department of Computer Science, University of Virginia, June 1998.

[18] M.A. Humphrey, F.C. Knabe, A.J. Ferrari and A.S. Grimshaw, “Accountability and Control of Process Creation in Metasystems,” In *Proceedings of the 2000 Network and Distributed Systems Security Conference* (NDSS’00), San Diego, CA, February 2000.

[19] L.J. Jin and A.S. Grimshaw, “From MetaComputing to Metabusiness Processing,” In *Proceedings IEEE International Conference on Cluster Computing -- Cluster 2000*, Saxony, Germany, December, 2000.

[20] S. Mullender ed., *Distributed Systems*, ACM Press, 1989

[21] A. Natrajan, M.A. Humphrey and A.S. Grimshaw, “Capacity and Capability Computing using Legion,” In *Proceedings of the 2001 International Conference on Computational Science*, San Francisco, CA, May 2001.

[22] A. Natrajan, A.J. Fox, M.A. Humphrey, A.S. Grimshaw, M. Crowley, N. Wilkins-Diehr, “Protein Folding on the Grid: Experiences using CHARMM under Legion on NPACI Resources,” *10th International Symposium on High Performance Distributed Computing (HPDC),* San Francisco, California, August 7-9, 2001.

[23] A. Nguyen-Tuong *et al.*, “Exploiting Data-Flow for Fault-Tolerance in a Wide-Area Parallel System”, *Proceedings of the 15th International Symposium on Reliable and Distributed Systems (SRDS-15)*, pp. 2-11, 1996.

[24] A. Nguyen-Tuong, S.J. Chapin, A.S. Grimshaw and C. Viles, “Using Reflection for Flexibility and Extensibility in a Metacomputing Environment,” *Technical Report 98-33*, University of Virginia, Dept. of Computer Science, November 19, 1998.

[25] L. Smarr and C.E. Catlett, “Metacomputing,” *Communications of the ACM.* 35(6):44-52, June 1992.

[26] B.S. White, M.P. Walker, M.A. Humphrey and A.S. Grimshaw “LegionFS: A Secure and Scalable File System Supporting Cross-Domain High-Performance Applications”, In *Proceedings of SuperComputing 2001*, Denver, CO. www.sc2001.org/papers/pap.pap324.pdf

[27] C.L. Viles *et al.*, **“**Enabling Flexibility in the Legion Run-Time Library,” *International Conference on Parallel and Distributed Processing Techniques (PDPTA ‘97)*, Las Vegas, NV, 1997.